Filter / Text Processing Commands

grep, awk, sed

grep

The grep utility is used to search for generalized regular expressions occurring in Linux files. Regular expressions, such as those shown above, are best specified in apostrophes (or single quotes) when specified in the grep utility. The egrep utility provides searching capability using an extended set of meta-characters. The syntax of the grep utility, some of the available options, and a few examples are shown below.

Syntax

grep [options] regexp [file[s]]

Common Options

-i	ignore case
-c	report only a count of the number of lines containing matches, not the matches themselves
-v	invert the search, displaying only lines that do not match
-n	display the line number along with the line on which a match was found
-s	work silently, reporting only the final status: 0, for match(es) found 1, for no matches 2, for errors
-l	list filenames, but not lines, in which matches were found

Examples

Consider the following file:

```
cat num.list
1 15 fifteen
2 14 fourteen
3 13 thirteen
4 12 twelve
5 11 eleven
6 10 ten
8 8 eight
9 7 seven
10 6 six
11 5 five
14 2 two
15 1 one
```

Here are some grep examples using this file. In the first we'll search for the number 15:

```
> grep '15' num.list
1 15 fifteen
15 1 one
```

Now we'll use the "-c" option to count the number of lines matching the search criterion:

```
> grep -c '15' num.list
    2
```

Here we'll be a little more general in our search, selecting for all lines containing the character 1 followed by either of 1, 2 or 5:

```
> grep '1[125]' num.list
1 15 fifteen
4 12 twelve
5 11 eleven
11 5 five
12 4 four
15 1 one
```

Now we'll search for all lines that begin with a space:

```
> grep '^ ' num.list
1 15 fifteen
2 14 fourteen
3 13 thirteen
4 12 twelve
5 11 eleven
6 10 ten
7 9 nine
8 8 eight
9 7 seven
```

Or all lines that don't begin with a space:

```
> grep '^[^ ]' num.list
10 6 six
11 5 five
12 4 four
13 3 three
14 2 two
15 1 one
```

The latter could also be done by using the -v option with the original search string, e.g.:

```
> grep -v '^ ' num.list
```

10 6 six
11 5 five
12 4 four
13 3 three
14 2 two
15 1 one

Here we search for all lines that begin with the characters 1 through 9:

```
> grep '^[1-9]' num.list
10 6 six
11 5 five
12 4 four
13 3 three
14 2 two
15 1 one
```

This example will search for any instances of t followed by zero or more occurrences of e:

```
> grep 'te*' num.list
1 15 fifteen
2 14 fourteen
3 13 thirteen
4 12 twelve
6 10 ten
8 8 eight
13 3 three
14 2 two
```

This example will search for any instances of t followed by one or more occurrences of e:

```
> grep 'tee*' num.list
1 15 fifteen
2 14 fourteen
3 13 thirteen
6 10 ten
```

We can also take our input from a program, rather than a file. Here we report on any lines output by the who program that begin with the letter l.

```
> who | grep '^l'
lcondron ttyp0 Dec 1 02:41 (lcondron-pc.acs.)
```

sed

The non-interactive, stream editor, sed, edits the input stream, line by line, making the specified changes, and sends the result to standard output.

Syntax

sed [options] edit_command [file]

The format for the editing commands are:

[address1[,address2]][function][arguments]

where the addresses are optional and can be separated from the function by spaces or tabs. The function is required. The arguments may be optional or required, depending on the function in use.

Line-number Addresses are decimal line numbers, starting from the first input line and incremented by one for each. If multiple input files are given the counter continues cumulatively through the files. The last input line can be specified with the "$" character.

Context Addresses are the regular expression patterns enclosed in slashes (/).

Commands can have 0, 1, or 2 comma-separated addresses with the following affects:

# of addresses	lines affected
0	every line of input
1	only lines matching the address
2	first line matching the first address and all lines until, and including, the line matching the second address. The process is then repeated on subsequent lines.

Substitution functions allow context searches and are specified in the form:

s/regular_expression_pattern/replacement_string/flag

and should be quoted with single quotes (’) if additional options or functions are specified. These patterns are identical to context addresses, except that while they are normally enclosed in slashes (/), any normal character is allowed to function as the delimiter, other than <space> and <newline>. The replacement string is not a regular expression pattern; characters do not have special meanings here, except:

&	substitute the string specified by regular_expression_pattern
\n	substitute the nth string matched by regular_expression_pattern enclosed in ’\(’, ’\)’ pairs.

These special characters can be escaped with a backslash (\) to remove their special meaning

Common Options

-e	script edit script
-n	don't print the default output, but only those lines specified by p or s///p functions
-f	script_file take the edit scripts from the file, script_file

Valid flags on the substitution functions include:

d	delete the pattern
g	globally substitute the pattern
p	print the line

Examples

This example changes all incidents of a comma (,) into a comma followed by a space (,) when doing output:

```
% cat filey | sed s/,/,\ /g
```

The following example removes all incidents of Jr preceded by a space (Jr) in filey:

```
% cat filey | sed s/\ Jr//g
```

To perform multiple operations on the input precede each operation with the -e (edit) option and quote the strings. For example, to filter for lines containing "Date: " and "From: " and replace these without the colon (:), try:

```
sed -e 's/Date: /Date /' -e 's/From: /From /'
```

To print only those lines of the file from the one beginning with "Date:" up to, and including, the one beginning with "Name:" try:

```
sed -n '/^Date:/,/^Name:/p'
```

To print only the first 10 lines of the input (a replacement for head):

```
sed -n 1,10p
```

awk, nawk, gawk

awk is a pattern scanning and processing language. Its name comes from the last initials of the three authors: Alfred. V. Aho, Brian. W. Kernighan, and Peter. J. Weinberger. nawk is new awk, a newer version of the program, and gawk is gnu awk, from the Free Software Foundation. Each version is a little different. Here we'll confine ourselves to simple examples which should be the same for all versions. On some OSs awk is really nawk.

awk searches its input for patterns and performs the specified operation on each line, or fields of the line, that contain those patterns. You can specify the pattern matching statements for awk either on

the command line, or by putting them in a file and using the -f program_file option.

Syntax

awk program [file]

where program is composed of one or more:

pattern { action }

fields. Each input line is checked for a pattern match with the indicated action being taken on a match. This continues through the full sequence of patterns, then the next line of input is checked.

Input is divided into records and fields. The default record separator is <newline>, and the variable NR keeps the record count. The default field separator is whitespace, spaces and tabs, and the variable NF keeps the field count. Input field, FS, and record, RS, separators can be set at any time to match any single character. Output field, OFS, and record, ORS, separators can also be changed to any single character, as desired. $n, where n is an integer, is used to represent the nth field of the input record, while $0 represents the entire input record.

BEGIN and END are special patterns matching the beginning of input, before the first field is read, and the end of input, after the last field is read, respectively.

Printing is allowed through the print, and formatted print, printf, statements.

Patterns may be regular expressions, arithmetic relational expressions, string-valued expressions, and boolean combinations of any of these. For the latter the patterns can be combined with the boolean operators below, using parentheses to define the combination:

||	or
&&	and
!	not

Comma separated patterns define the range for which the pattern is applicable, e.g.:

/first/,/last/

selects all lines starting with the one containing first, and continuing inclusively, through the one containing last.

To select lines 15 through 20 use the pattern range:

NR == 15, NR == 20

Regular expressions must be enclosed with slashes (/) and meta-characters can be escaped with the backslash (\). Regular expressions can be grouped with the operators:

|	or, to separate alternatives

+	one or more
?	zero or one

A regular expression match can be either of:

~	contains the expression
!~	does not contain the expression

So the program:

```
$1 ~ /[Ff]rank/
```

is true if the first field, $1, contains "Frank" or "frank" anywhere within the field. To match a field identical to "Frank" or "frank" use:

```
$1 ~ /^[Ff]rank$/
```

Relational expressions are allowed using the relational operators:

<	less than
<=	less than or equal to
==	equal to
>=	greater than or equal to
!=	not equal to
>	greater than

Offhand you don’t know if variables are strings or numbers. If neither operand is known to be numeric, than string comparisons are performed. Otherwise, a numeric comparison is done. In the absence of any information to the contrary, a string comparison is done, so that:

```
$1 > $2
```

will compare the string values. To ensure a numerical comparison do something similar to:

```
( $1 + 0 ) > $2
```

The mathematical functions: exp, log and sqrt are built-in

Some other built-in functions include:

index(s,t)	returns the position of string s where t first occurs, or 0 if it doesn’t
length(s)	returns the length of string s
substr(s,m,n)	returns the n-character substring of s, beginning at position m

Arrays are declared automatically when they are used, e.g.:

```
arr[i] = $1
```

assigns the first field of the current input record to the ith element of the array.

Flow control statements using if-else, while, and for are allowed with C type syntax:

```
for (i=1; i <= NF; i++) {actions}
```

while (i<=NF) {actions}
if (i<NF) {actions}

Common Options

-f program_file read the commands from program_file
-Fc use character c as the field separator character

Examples

% cat filex | tr a-z A-Z | awk -F: '{printf ("7R %-6s %-9s %-24s \n",$1,$2,$3)}'>upload.file

cats filex, which is formatted as follows:

nfb791:99999999:smith
7ax791:999999999:jones
8ab792:99999999:chen
8aa791:999999999:mcnulty

changes all lower case characters to upper case with the tr utility, and formats the file into the following which is written into the file upload.file:

7R NFB791 99999999 SMITH
7R 7AX791 999999999 JONES
7R 8AB792 99999999 CHEN
7R 8AA791 999999999 MCNULTY

cut - select parts of a line

The ***cut*** command allows a portion of a file to be extracted for another use.

Syntax

```
cut [options] file
```

Common Options

-c character_list character positions to select (first character is 1)
-d delimiter field delimiter (defaults to <**TAB**>)
-f field_list fields to select (first field is 1)

Both the character and field lists may contain comma-separated or blank-character-separated numbers (in increasing order), and may contain a hyphen (**-**) to indicate a range. Any numbers missing at either before (e.g. -5) or after (e.g. 5-) the hyphen indicates the full range starting with the first, or ending with the last character or field, respectively. Blank-character-separated lists must be enclosed in quotes. The field delimiter should be enclosed in quotes if it has special meaning to the shell, e.g. when specifying a <**space**> or **<TAB>** character.

Examples

In these examples we will use the file **users**:

```
jdoe John Doe 4/15/96
lsmith Laura Smith 3/12/96
```

```
pchen Paul Chen 1/5/96
jhsu Jake Hsu 4/17/96
sphilip Sue Phillip 4/2/96
```

If you only wanted the username and the user's real name, the ***cut*** command could be used to get only that information:

```
% cut -f 1,2 users
jdoe John Doe
lsmith Laura Smith
pchen Paul Chen
jhsu Jake Hsu
sphilip Sue Phillip
```

The ***cut*** command can also be used with other options. The **-c** option allows characters to be the selected cut. To select the first 4 characters:

```
% cut -c 1-4 users
This yields:
jdoe
lsmi
pche
jhsu
sphi
```

thus cutting out only the first 4 characters of each line.

paste - merge files

The ***paste*** command allows two files to be combined side-by-side. The default delimiter between the columns in a paste is a tab, but options allow other delimiters to be used.

Syntax
`paste [options] file1 file2`

Common Options
-d list list of delimiting characters
-s concatenate lines
The list of **delimiters** may include a single character such as a comma; a quoted string, such as a space; or any of the following escape sequences:

```
\n <newline> character
\t <tab> character
\\ backslash character
\0 empty string (non-null character)
```

It may be necessary to quote delimiters with special meaning to the shell.
A hyphen (**-**) in place of a file name is used to indicate that field should come from standard input.

Examples
Given the file **users**:

```
jdoe John Doe 4/15/96
lsmith Laura Smith 3/12/96
pchen Paul Chen 1/5/96
jhsu Jake Hsu 4/17/96
sphilip Sue Phillip 4/2/96
```
and the file **phone:**
```
John Doe 555-6634
Laura Smith 555-3382
Paul Chen 555-0987
Jake Hsu 555-1235
Sue Phillip 555-7623
```

the ***paste*** command can be used in conjunction with the ***cut*** command to create a new file, **listing**, that includes the username, real name, last login, and phone number of all the users. First, extract the phone numbers into a temporary file, **temp.file**:

```
% cut -f2 phone > temp.file
555-6634
555-3382
555-0987
555-1235
555-7623
```

The result can then be pasted to the end of each line in **users** and directed to the new file, **listing**:

```
% paste users temp.file > listing
jdoe John Doe 4/15/96 237-6634
lsmith Laura Smith 3/12/96 878-3382
pchen Paul Chen 1/5/96 888-0987
jhsu Jake Hsu 4/17/96 545-1235
sphilip Sue Phillip 4/2/96 656-7623
```

This could also have been done on one line without the temporary file as:

```
% cut -f2 phone | paste users - > listing
```

with the same results. In this case the hyphen (**-**) is acting as a placeholder for an input field (namely, the output of the ***cut*** command).

sort - sort file contents

The ***sort*** command is used to order the lines of a file. Various options can be used to choose the order as well as the field on which a file is sorted. Without any options, the sort compares entire lines in the file and outputs them in ASCII order (numbers first, upper case letters, then lower case letters).

Syntax

sort [options] [+pos1 [-pos2]] file

Common Options

-b ignore leading blanks (<**space**> & <**tab**>) when determining starting and ending characters for the sort key
-d dictionary order, only letters, digits, <**space**> and <**tab**> are significant
-f fold upper case to lower case

-k keydef sort on the defined keys (not available on all systems)
-i ignore non-printable characters
-n numeric sort
-o outfile output file
-r reverse the sort
-t char use char as the field separator character
-u unique; omit multiple copies of the same line (after the sort)
+pos1 [**-**pos2] (old style) provides functionality similar to the "-k keydef" option.

For the **+/-position** entries **pos1** is the starting word number, beginning with **0** and **pos2** is the ending word number. When **-pos2** is omitted the sort field continues through the end of the line. Both pos1 and **pos2** can be written in the form **w.c**, where **w** is the word number and **c** is the character within the word. For **c 0** specifies the delimiter preceding the first character, and **1** is the first character of the word. These entries can be followed by type modifiers, e.g. **n** for numeric, **b** to skip blanks, etc.

The **keydef** field of the "**-k**" option has the syntax:
`start_field [type] [ ,end_field [type] ]`

where:
`start_field, end_field` define the keys to restrict the sort to a portion of the line
type modifies the sort, valid modifiers are given the single characters (bdfiMnr)
from the similar sort options, e.g. a type **b** is equivalent to "**-b**", but applies
only to the specified field

Examples
In the file **users**:
```
jdoe John Doe 4/15/96
lsmith Laura Smith 3/12/96
pchen Paul Chen 1/5/96
jhsu Jake Hsu 4/17/96
sphilip Sue Phillip 4/2/96
sort users yields the following:
jdoe John Doe 4/15/96
jhsu Jake Hsu 4/17/96
lsmith Laura Smith 3/12/96
pchen Paul Chen 1/5/96
sphilip Sue Phillip 4/2/96
```

If, however, a listing sorted by last name is desired, use the option to specify which field to sort on (fields are numbered starting at 0):
```
% sort +2 users:
pchen Paul Chen 1/5/96
jdoe John Doe 4/15/96
jhsu Jake Hsu 4/17/96
sphilip Sue Phillip 4/2/96
lsmith Laura Smith 3/12/96
```

To sort in reverse order:
```
% sort -r users:
sphilip Sue Phillip 4/2/96
pchen Paul Chen 1/5/96
lsmith Laura Smith 3/12/96
```

```
jhsu Jake Hsu 4/17/96
jdoe John Doe 4/15/96
```

A particularly useful ***sort*** option is the **-u** option, which eliminates any duplicate entries in a file while ordering the file. For example, the file todays.logins:

```
sphillip
jchen
jdoe
lkeres
jmarsch
ageorge
lkeres
proy
jchen
```

shows a listing of each username that logged into the system today. If we want to know how many unique users logged into the system today, using sort with the **-u** option will list each user only once. (The command can then be piped into "***wc -l***" to get a number):

```
% sort -u todays.logins
ageorge
jchen
jdoe
jmarsch
lkeres
proy
sphillip
```

uniq - remove duplicate lines

uniq filters duplicate adjacent lines from a file.

Syntax
uniq [options] [+|-n] file [file.new]

Common Options
-d one copy of only the repeated lines
-u select only the lines not repeated
+n ignore the first **n characters**
-s n same as above (SVR4 only)
-n skip the first **n fields**, including any blanks (<**space**> & <**tab**>)
-f fields same as above (SVR4 only)

Examples
Consider the following file and example, in which ***uniq*** removes the 4th line from **file** and places the result in a file called **file.new**.

```
$ cat file
```

```
1 2 3 6
4 5 3 6
7 8 9 0
7 8 9 0

$ uniq file file.new

$ cat file.new
1 2 3 6
4 5 3 6
7 8 9 0
```

Below, the **-n** option of the ***uniq*** command is used to skip the first 2 fields in **file**, and filter out lines which are duplicates from the 3rd field onward.

```
$ uniq -2 file
1 2 3 6
7 8 9 0
```

tee - copy command output

tee sends standard in to specified files and also to standard out. It's often used in command pipelines.

Syntax

```
tee [options] [file[s]]
```

Common Options

-a	append the output to the files
-i	ignore interrupts

Examples

In this first example the output of who is displayed on the screen and stored in the file users.file:

```
> who | tee users.file
     condron     ttyp0      Apr 22 14:10      (lcondron-pc.acs.)
     frank       ttyp1      Apr 22 16:19      (nyssa)
     condron     ttyp9      Apr 22 15:52      (lcondron-mac.acs)

> cat users.file
     Condron     ttyp0      Apr 22 14:10      (lcondron-pc.acs.)
     Frank       ttyp1      Apr 22 16:19      (nyssa)
     Condron     ttyp9      Apr 22 15:52      (lcondron-mac.acs)
```

In this next example the output of who is sent to the files users.a and users.b. It is also piped to the wc command, which reports the line count.

```
> who | tee users.a users.b | wc -l
     3

> cat users.a
     condron     ttyp0      Apr 22 14:10      (lcondron-pc.acs.)
     frank       ttyp1      Apr 22 16:19      (nyssa)
```

```
     condron     ttyp9       Apr 22 15:52      (lcondron-mac.acs)

> cat users.b
     condron     ttyp0       Apr 22 14:10      (lcondron-pc.acs.)
     frank       ttyp1       Apr 22 16:19      (nyssa)
     condron     ttyp9       Apr 22 15:52      (lcondron-mac.acs)
```

In the following example a long directory listing is sent to the file files.long. It is also piped to the grep command which reports which files were last modified in August.

```
> ls -l | tee files.long |grep Aug
      1 drwxr-sr-x 2 condron 512 Aug 8 1995 News/
      2 -rw-r--r-- 1 condron 1076 Aug 8 1995 magnus.cshrc
      2 -rw-r--r-- 1 condron 1252 Aug 8 1995 magnus.login

> cat files.long
total 34
      2 -rw-r--r-- 1 condron 1253 Oct 10 1995 #.login#
      1 drwx------ 2 condron 512 Oct 17 1995 Mail/
      1 drwxr-sr-x 2 condron 512 Aug 8 1995 News/
      5 -rw-r--r-- 1 condron 4299 Apr 21 00:18 editors.txt
      2 -rw-r--r-- 1 condron 1076 Aug 8 1995 magnus.cshrc
      2 -rw-r--r-- 1 condron 1252 Aug 8 1995 magnus.login
      7 -rw-r--r-- 1 condron 6436 Apr 21 23:50 resources.txt
      4 -rw-r--r-- 1 condron 3094 Apr 18 18:24 telnet.ftp
      1 drwxr-sr-x 2 condron 512 Apr 21 23:56 uc/
      1 -rw-r--r-- 1 condron 1002 Apr 22 00:14 uniq.tee.txt
      1 -rw-r--r-- 1 condron 1001 Apr 20 15:05 uniq.tee.txt~
      7 -rw-r--r-- 1 condron 6194 Apr 15 20:18 Linuxgrep.txt
```